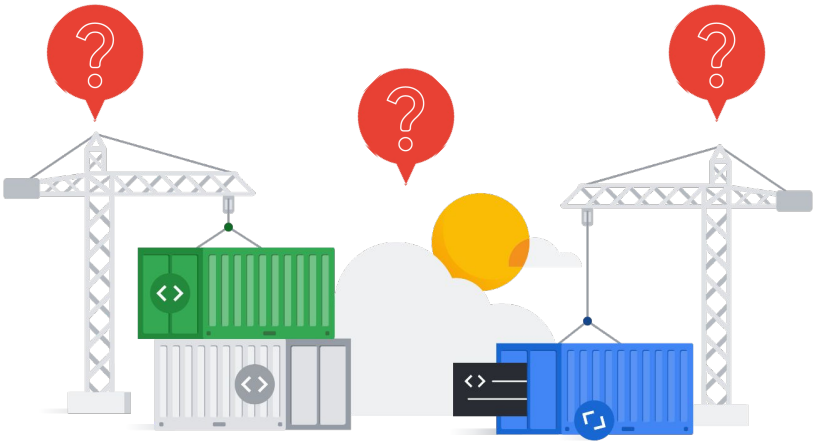

# Introduction to Containers and Kubernetes

## What are containers? How do they work?





Containerization helps development teams move fast, deploy software efficiently, and operate at an unprecedented scale. But what exactly are containers, how do they work, and where does Kubernetes come into play?

The goal of this second section of the course was designed to help answer those questions.

# Introduction to Containers and Kubernetes

01 Containers

02 Container images

03 Kubernetes

04 Google Kubernetes Engine



You'll explore:

- Containers
- Container images
- Kubernetes
- Google Kubernetes Engine

You'll also get hands-on practice with Cloud Build.

# Introduction to Containers and Kubernetes

01 Containers

02 Container images

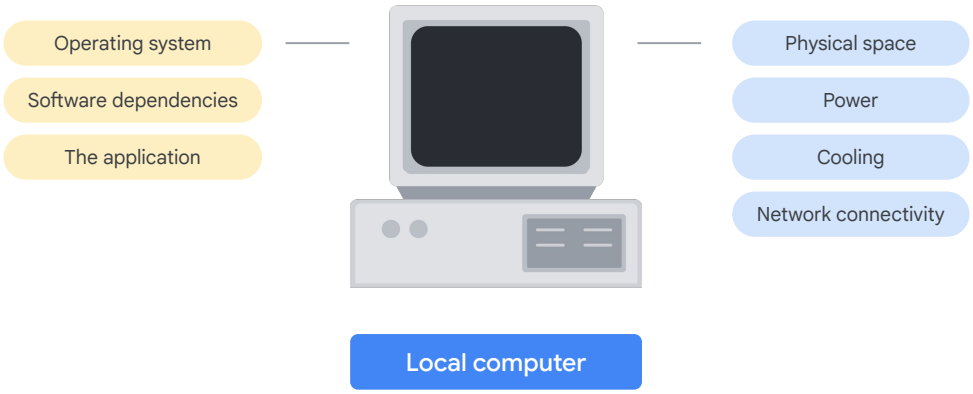
03 Kubernetes

04 Google Kubernetes Engine



So, what is a container? To answer that question, we need to first explore how applications are deployed.

## Not long ago, the common way to deploy an application was on a local computer





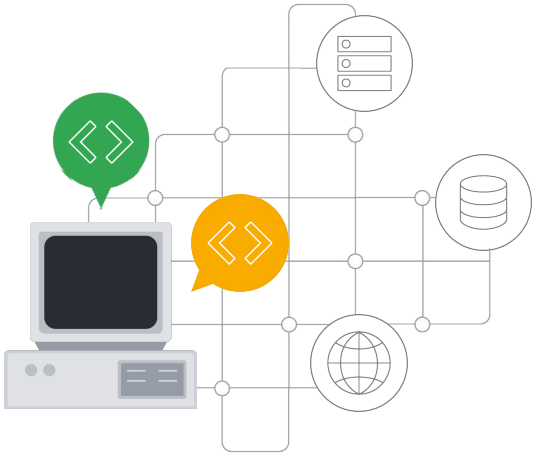

Not long ago, the common way to deploy an application was on a local computer. To set one up, you needed physical space, power, cooling, and network connectivity. Then you needed to install an operating system, any software dependencies, and finally, the application.

When you needed more processing power, redundancy, security, or scalability, you'd add more computers. And it was common for each computer to have a single purpose, for example, for a database, web server, or content delivery.

This practice wasted resources and took a lot of time to deploy, maintain, and scale.

## Then came virtualization

Virtualization is the process of creating a virtual version of a physical resource, such as a server, storage device, or network.

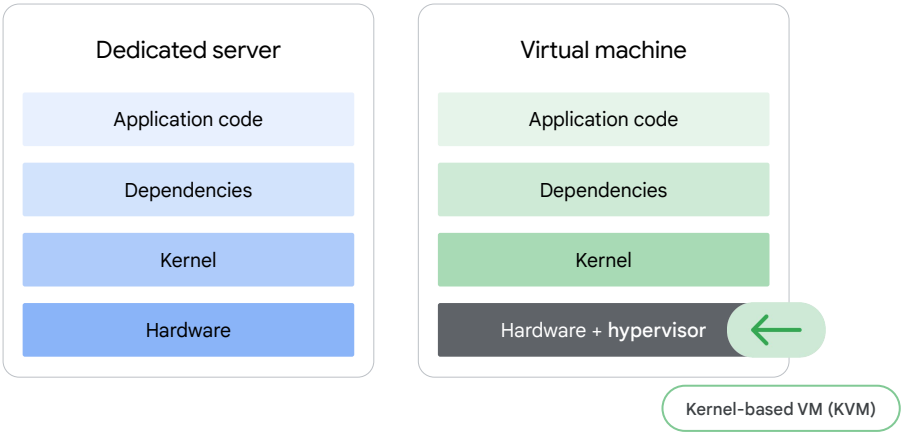




Then came virtualization, which is the process of creating a virtual version of a physical resource, such as a server, storage device, or network.

Virtualization made it possible to run multiple virtual servers and operating systems on one local computer.

# Hypervisor

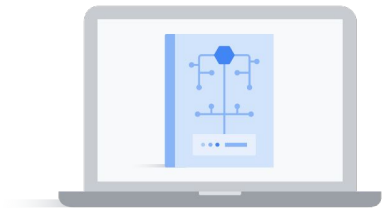





The software layer that breaks the dependencies of an operating system on the underlying hardware and allows several virtual machines to share that hardware is called a *hypervisor*.

Kernel-based Virtual Machine, or KVM, is one well-known hypervisor. Today, you can use virtualization to deploy new servers fairly quickly.

# Virtualization



- It takes less time to deploy new solutions.
- Fewer resources are wasted.
- Portability is improved.
- It isn't easy to move a VM from one hypervisor.
- VM operating systems take time to boot up.

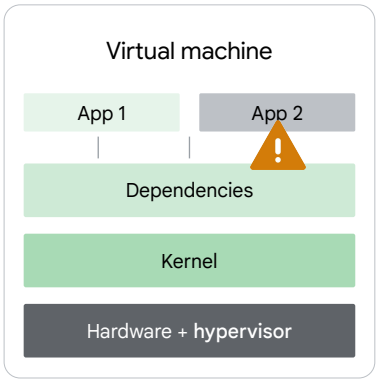


With virtualization, it takes less time to deploy new solutions. Fewer resources are wasted, and portability is improved because virtual machines can be imaged and easily moved.

However, an application, all its dependencies, and operating system are still bundled together.

It's not easy to move a VM from one hypervisor product to another, and every time you start a VM, its operating system takes time to boot up.

## Applications that share dependencies are not isolated from each other




Try and solve this problem with:

- Software engineering policies
  - Need to be updated occasionally.
- Integration tests
  - Can cause novel failure modes that are hard to troubleshoot.
  - Slow down development.

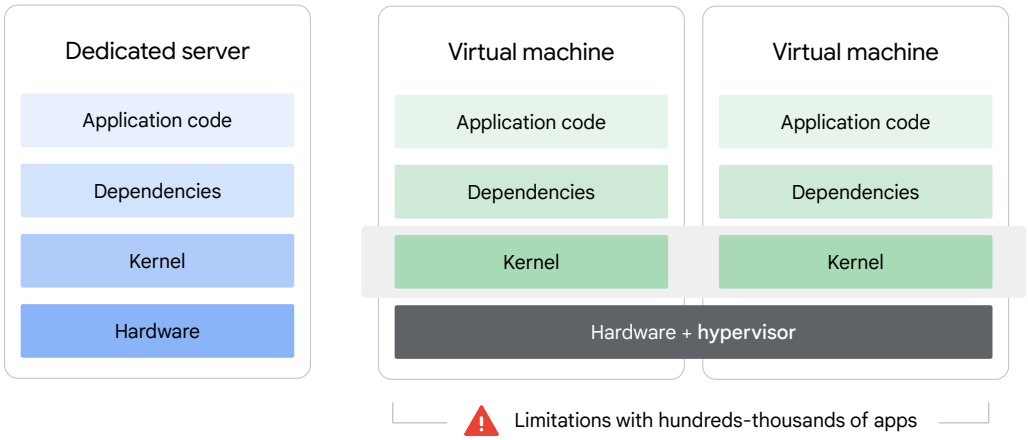


But running multiple applications within a single VM creates another problem: applications that share dependencies are not isolated from each other. The resource requirements of one application can starve other applications of the resources they need. Also, a dependency upgrade for one application might cause another to stop working.

You can try to solve this problem with rigorous *software engineering policies*. For example, you can lock down the dependencies so that no application is allowed to make changes; but this can lead to new problems because dependencies need to be upgraded occasionally.

You can also add *integration tests* to ensure that applications work as intended. However, dependency problems can cause novel failure modes that are hard to troubleshoot. Also, they significantly slow down development if you have to rely on integration tests to confirm the basic integrity of your application environment.

## VM-centric solution: Run a dedicated VM for each application

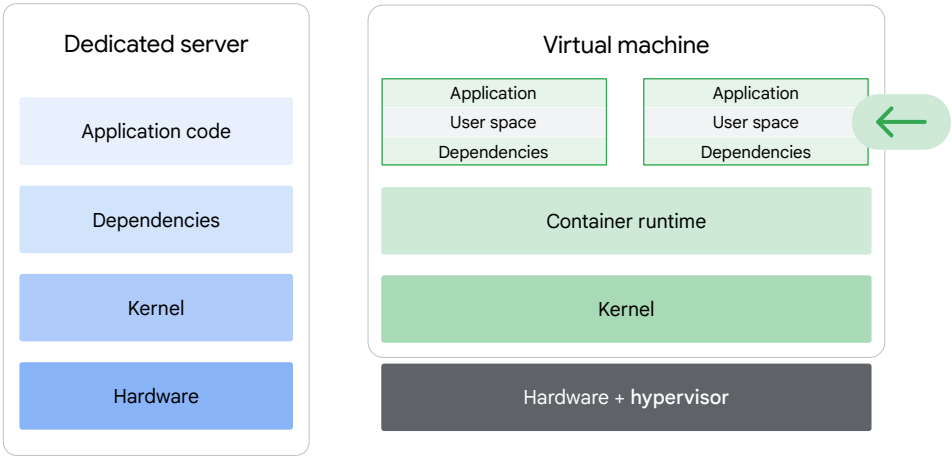





The VM-centric way to solve this problem is to run a dedicated virtual machine for *each* application.

Each application maintains its own dependencies, and the kernel is isolated so one application won't affect the performance of another. The result is that two complete copies of the kernel are running.
Scale this to hundreds or thousands of applications and you see its limitations

## Implement abstraction at the level of the application and its dependencies

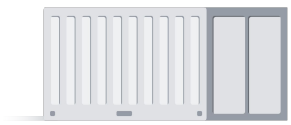





A more efficient way to resolve the dependency problem is to implement abstraction at the level of the *application and its dependencies*. You don't have to virtualize the entire machine, or even the entire operating system, just the user space.

The **user space** is all the code that resides above the kernel, and it includes applications and their dependencies.

## Containers are isolated user spaces for running application code



Containers are lightweight because they don't carry a full operating system.

Containers can be scheduled or integrated tightly with the underlying system.

Containers can be created and shut down quickly.

Containers do not boot an entire VM or initialize an operating system for each application.

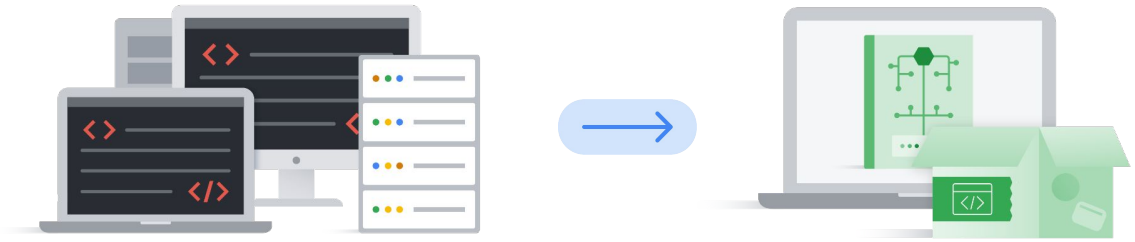


This is what it means to create containers. **Containers** are isolated user spaces for running application code.

Containers:

- Are lightweight because they don't carry a full operating system.
- Can be scheduled or integrated tightly with the underlying system, which is efficient.
- And can be created and shut down quickly, because they just start and stop operating system processes, and don't boot an entire VM or initialize an operating system for each application.

## Develop containers in the usual ways, execute on VMs



Develop application code on desktops, laptops, and servers

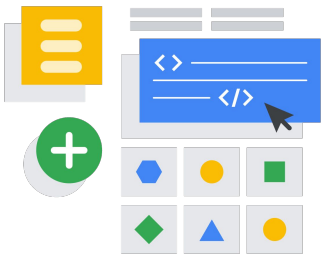
The container can execute final code on VMs



With containers, you can still develop application code in the usual ways–on desktops, laptops, and servers.

However, the container can execute final code on VMs. The application code is packaged with all the dependencies it needs, and the engine that executes the container is responsible for making them available at runtime.

## What makes containers so appealing to developers?



- They're a code-centric way to deliver high-performing, scalable applications.
- They provide access to reliable underlying hardware and software.
- Code will run successfully regardless if it is on a local machine or in production.
- They make it easier to build applications that use the microservices design pattern.



But what makes containers so appealing to developers?

- First, they're a code-centric way to deliver high-performing, scalable applications.
- Second, containers provide access to reliable underlying hardware and software.
- With a Linux kernel base, developers can be confident that code will run successfully regardless if it is on a local machine or in production. And if incremental changes are made to a container based on a production image, it can be deployed quickly with a single file copy. This accelerates development.
- And finally, containers make it easier to build applications that use the microservices design pattern, that is, with loosely coupled, fine-grained components. This modular design pattern allows the operating system to scale and upgrade components of an application without affecting the application as a whole.

# Introduction to Containers and Kubernetes

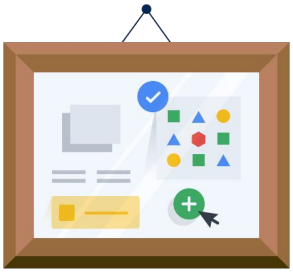
01 Containers

02 Container images

03 Kubernetes

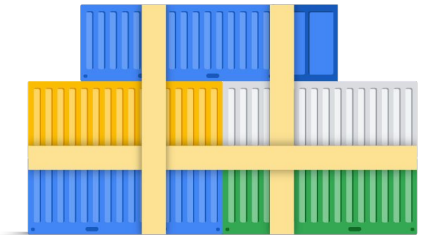
04 Google Kubernetes Engine

# Container images



Application + dependencies =

Container image



Developers can package and ship an application without worrying about the system that it will run on.



An application and its dependencies are called an *image*, and a container is simply a running instance of an image. By building software into **container images**, developers can package and ship an application without worrying about the system it will run on.

## You need software to build and run container images

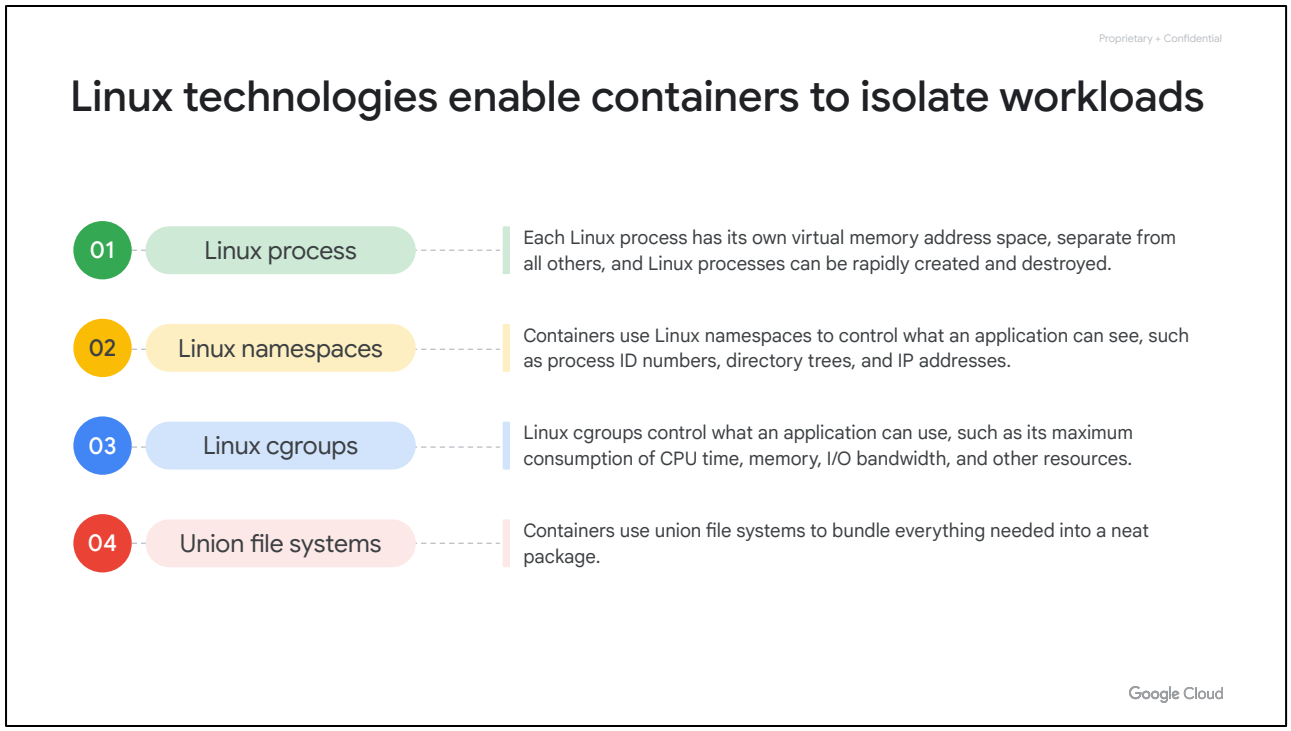
Docker

- Docker is an open-source technology.
- It can be used to create and run applications in containers.
- It doesn't orchestrate applications at scale.



One option is **Docker**. Although this open-source technology can be used to create and run applications in containers, it doesn't offer a way to *orchestrate* those applications at scale like Kubernetes does.

Later in this course, you'll use Google's **Cloud Build** to create Docker-formatted container images. But to build and run container images, you need software.

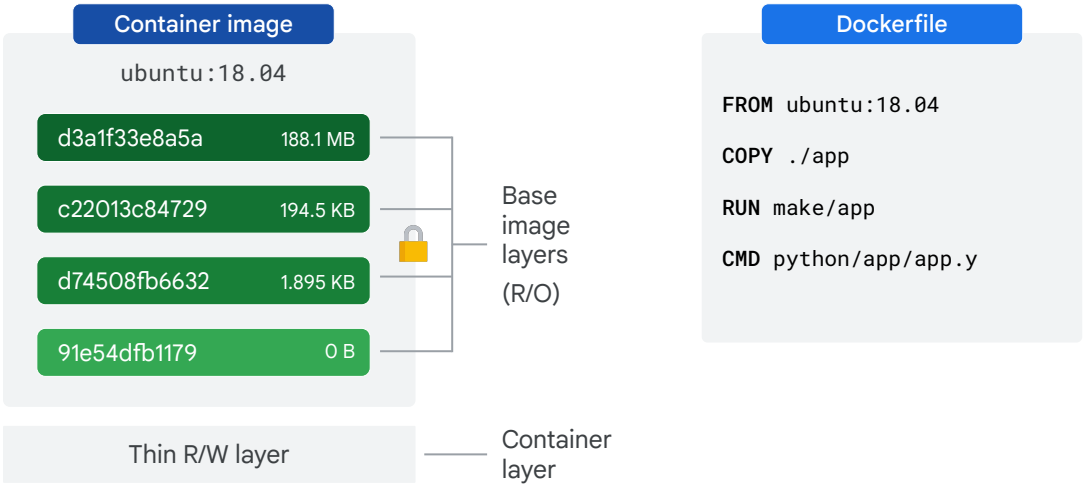



A container has the power to isolate workloads, and this ability comes from a combination of several Linux technologies.

- The first is the foundation of the **Linux process**. Each Linux process has its own virtual memory address space, separate from all others, and Linux processes can be rapidly created and destroyed.
- The next technology is **Linux namespaces**. Containers use Linux namespaces to control *what* an application can see, such as process ID numbers, directory trees, and IP addresses. It's important to note that Linux namespaces are *not* the same thing as Kubernetes namespaces, which you will learn more about in the third section of this course.
- The third technology is **Linux cgroups.** Linux cgroups control *what* an application can use, such as its maximum consumption of CPU time, memory, I/O bandwidth, and other resources.
- And finally, containers use **union file systems** to bundle everything needed into a neat package. This requires combining applications and their dependencies into a set of clean, minimal layers.

Let's explore how this works.

## The container manifest and Dockerfile

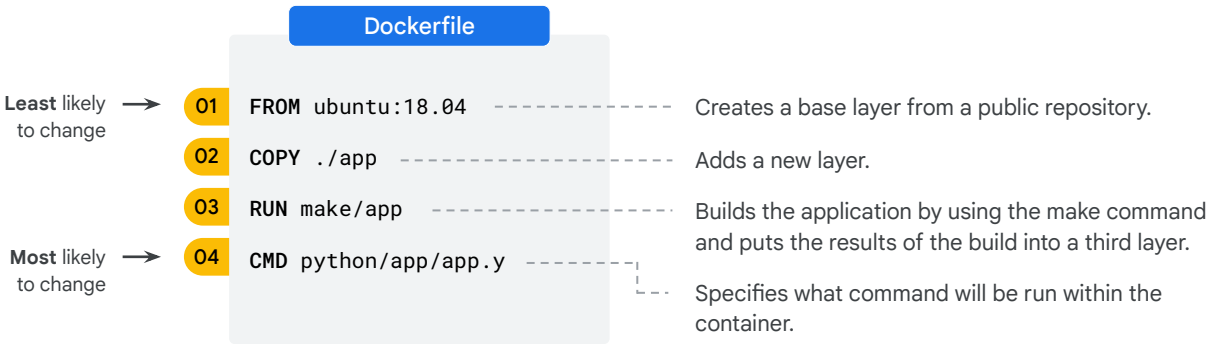





A container image is structured in layers, and the tool used to build the image reads instructions from a file called the *container manifest*.

For Docker-formatted container images, that's called a Dockerfile. Each instruction in the Dockerfile specifies a layer inside the container image. Each layer is read-only, but when a container runs from this image, it will also have a writable, ephemeral topmost layer.

## A Dockerfile contains four commands, each of which creates a layer

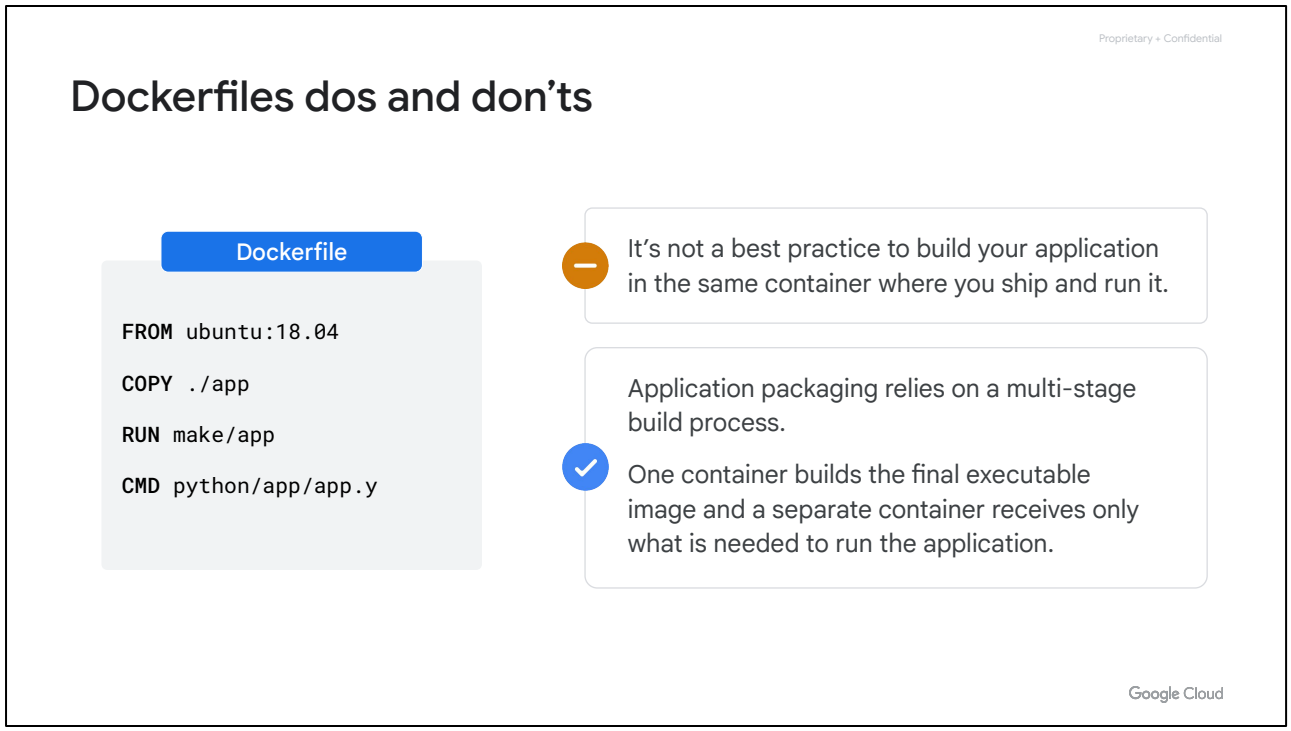





Let's explore a simple Dockerfile. A Dockerfile contains four commands, each of which creates a layer. For the purposes of this training, this Dockerfile has been a little oversimplified for modern use.

- The **FROM** statement starts by creating a base layer, which is pulled from a public repository. This base layer is the Ubuntu Linux runtime environment of a specific version.
- The **COPY** command adds a new layer, which contains some files copied in from your build tool's current directory.
- The **RUN** command builds the application by using the "make" command and puts the results of the build into a third layer.
- And finally, the last layer specifies what command you should run within the container when it is launched.

When you write a Dockerfile, the layers should start with those least likely to change at the top, and the layers most likely to change at the bottom.




So I mentioned this Dockerfile example is oversimplified. Let me explain what I meant.

- Currently, it’s not a best practice to build your application in the same container where you ship and run it. After all, your build tools are at best just clutter in a deployed container, and at worst they are an additional attack surface.
- Today, application packaging relies on a multi-stage build process, where one container builds the final executable image, and a separate container receives only what is needed to run the application.

## Dockerfiles dos and don’ts

Dockerfile

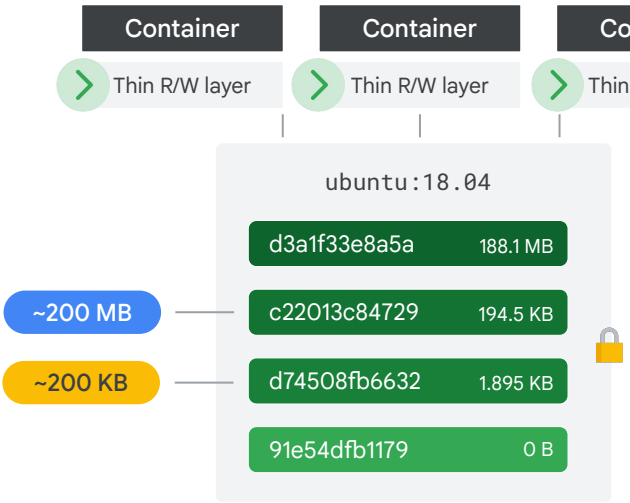
```
FROM ubuntu:18.04
COPY ./app
RUN make/app
CMD python/app/app.y
```

- The container runtime adds a new writable layer on top of the underlying layers called the **container layer**.
- All changes made to the running container are written to this thin writable container layer.
- The container layers is ephemeral.



- When you launch a new container from an image, the container runtime adds a new writable layer on top of the underlying layers. This layer is called the *container layer*.
- All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin, writable container layer. And they’re ephemeral, which means that when the container is deleted, the contents of this writable layer are lost forever. The underlying container image remains unchanged. So when it comes to application design, this means that permanent data must be stored somewhere *other* than a running container image.

## Each container has its own writable container layer






Because each container has its own writable container layer, and all changes are stored in this layer, multiple containers can share access to the same underlying image while still maintaining their own data state. This allows container images to get smaller with each layer.
For example, a base application image might be 200 MB, but the difference to the next point release might only be 200 KB.

When building a container, instead of copying the entire image, it creates a layer with just the difference. When a container is run, the container runtime pulls down the layers it needs. When a container is updated, only the difference needs to be copied. This is much faster than running a new virtual machine.

## How can you get or create containers?

Use publicly available open-source container images as the base for your own images.

Use the Google maintained **Artifact Registry** at pkg.dev, which contains public, open source images.

Use container images available in other public repositories, like the **Docker Hub Registry** and **GitLab.**



So, how can you get or create containers?

- It's common to use publicly available open-source container images as the base for your own images, or for unmodified use.
- Google maintains Artifact Registry at pkg.dev, which contains public, open source images. It also provides Google Cloud customers with a place to store their own container images and is integrated with Identity and Access Management (IAM). This allows storing container images that are private to your project.
- Container images are also available in other public repositories, like the Docker Hub Registry and GitLab.

## How can you get or create containers?

Use **Cloud Build**, a Google-provided managed service for building containers.

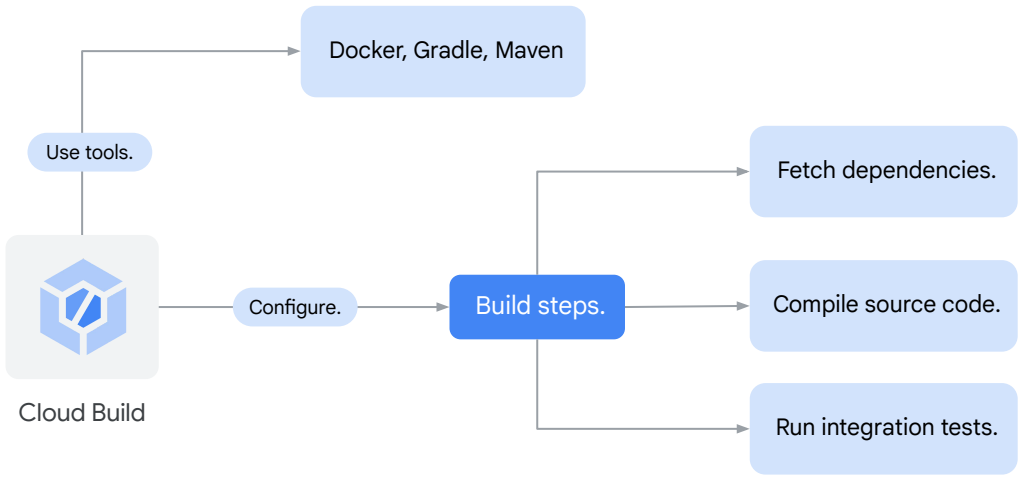
- Is integrated with Cloud IAM.
- Is designed to retrieve the source code builds from different code repositories:
  - Cloud Source Repositories
  - GitHub
  - Bitbucket



Google provides a managed service for building containers called **Cloud Build**.

Cloud Build is integrated with Cloud IAM and was designed to retrieve the source code builds from different code repositories, including Cloud Source Repositories, or git-compatible repositories like GitHub and Bitbucket.

## How to generate a build with Cloud Build

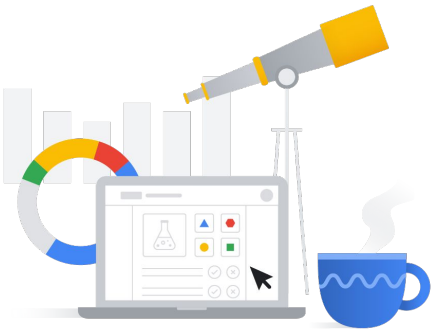





To generate a build with Cloud Build, you must define a series of steps. For example, you can configure build steps to fetch dependencies, compile source code, run integration tests, or use tools such as Docker, Gradle, and Maven. Each build step in Cloud Build runs in a Docker container.

From there, Cloud Build can deliver the newly built images to various execution environments including Google Kubernetes Engine, App Engine, and Cloud Run functions.

# Lab: Working with Cloud Build



01 Use provided code to build a Docker container image and a Dockerfile.

02 Upload the container to the Google Cloud Artifact Registry.



Now it's time to gain some hands-on experience with Cloud Build.

In the lab titled, "Working with Cloud Build," you'll use provided code to build a Docker container image and a Dockerfile.

From there, you'll upload the container to the Google Cloud Artifact Registry, which is a private Docker repository to securely store and manage Docker images.

# Introduction to Containers and Kubernetes

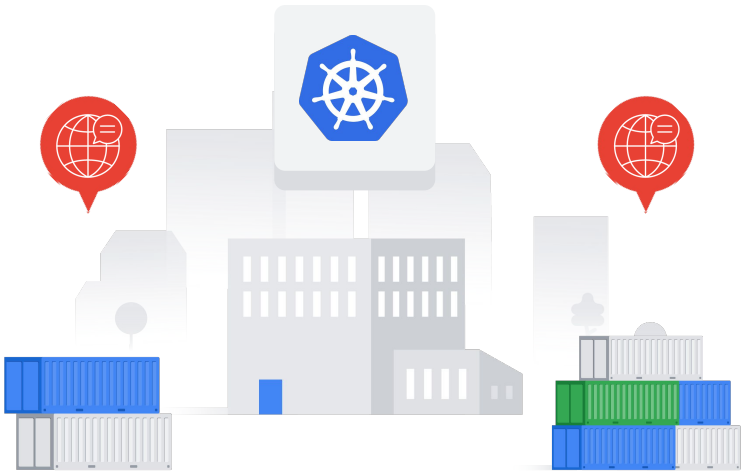
01 Containers

02 Container images

03 Kubernetes

04 Google Kubernetes Engine

## Kubernetes





So let's say that your organization has implemented containers, and because containers are so lean, your coworkers are creating them in numbers that exceed the counts of virtual machines you used to have.

Let's also say that the applications that run in the containers need to communicate over the network, but you don't have a network fabric that lets containers find each other.

Kubernetes can help.

## What is Kubernetes?

It's an open source platform for managing containerized workloads and services.

It makes it easy to orchestrate many containers on many hosts, scale them as microservices, and deploy rollouts and rollbacks.

It's a set of APIs to deploy containers on a set of nodes called a cluster.

It's divided into a set of primary components that run as the control plane and a set of nodes that run containers.

You can describe a set of applications and how they should interact with each other, and Kubernetes figures how to make that happen.



- So, what is Kubernetes? Kubernetes is an open source platform for managing containerized workloads and services.
- It makes it easy to orchestrate many containers on many hosts, scale them as microservices, and easily deploy rollouts and rollbacks.
- At the highest level, Kubernetes is a set of APIs that you can use to deploy containers on a set of nodes called a *cluster*.
- The system is divided into a set of primary components that run as the control plane and a set of nodes that run containers. In Kubernetes, a node represents a computing instance, like a machine. Note that this is different to a node on Google Cloud, which is a virtual machine that runs in Compute Engine.
- You can describe a set of applications and how they should interact with each other, and Kubernetes determines how to make that happen.

## Kubernetes supports declarative configurations


Describe the state of the infrastructure you want to achieve.

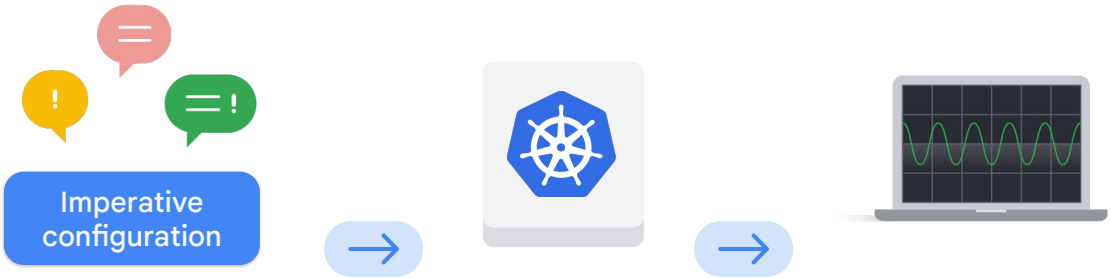
Kubernetes's job is to make the deployed system conform to your desired state.

Reduces the risk of error.



Kubernetes supports **declarative configurations**. When you administer your infrastructure declaratively, you describe the desired state you want to achieve, instead of issuing a series of commands to achieve that desired state. Kubernetes's job is to make the deployed system conform to your desired state and to keep it there in spite of failures. Declarative configuration saves you work. Because the system's desired state is always documented, it also reduces the risk of error.

## Kubernetes also allows imperative configurations



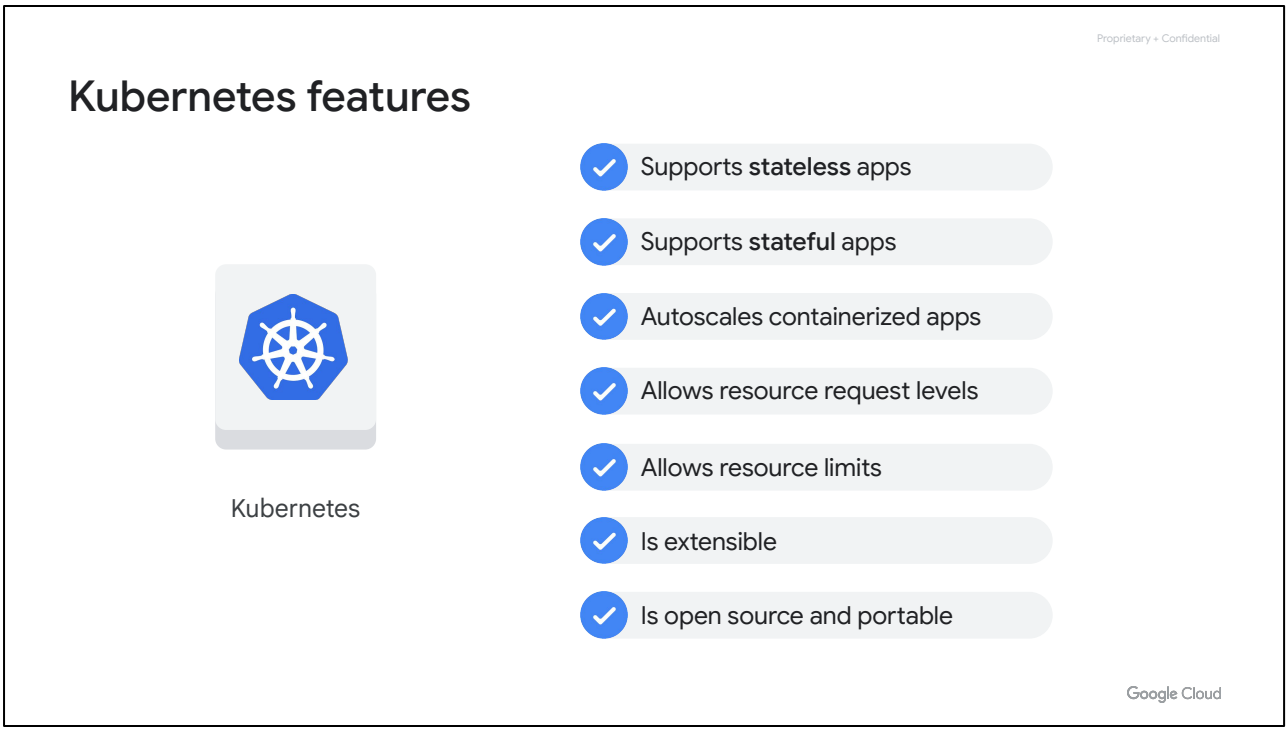

Issue commands to change the system's state.

Kubernetes automatically keeps a system in a state you declare.

Experienced admins use them for quick, temporary fixes.



Kubernetes also allows **imperative configuration**, in which you issue commands to change the system's state. One of the primary strengths of Kubernetes is its ability to automatically keep a system in a state you declare. Therefore, experienced Kubernetes administrators use imperative configuration only for quick temporary fixes and as a tool when building a declarative configuration.




Now that you have a better understanding of what Kubernetes is, let's explore some of its features.

- Kubernetes supports different workload types. It supports stateless applications, such as Nginx or Apache web servers.
- It also supports stateful applications where user and session data can be stored persistently. It also supports batch jobs and daemon tasks.
- Kubernetes can automatically scale containerized applications in and out based on resource utilization.
- Kubernetes allows users to specify resource request levels and resource limits for workloads. Resource controls help Kubernetes improve the overall workload performance within a cluster.
- Kubernetes is extensible through a rich ecosystem of plugins and addons. For example, Kubernetes Custom Resource Definitions let developers define new types of resources that can be created, managed, and used in Kubernetes.
- And finally, because it's open source, Kubernetes is portable and can be deployed anywhere–whether on premises or on another cloud service provider. This means that Kubernetes workloads can be moved freely without vendor lock-in.

# Introduction to Containers and Kubernetes

01 Containers

02 Container images

03 Kubernetes

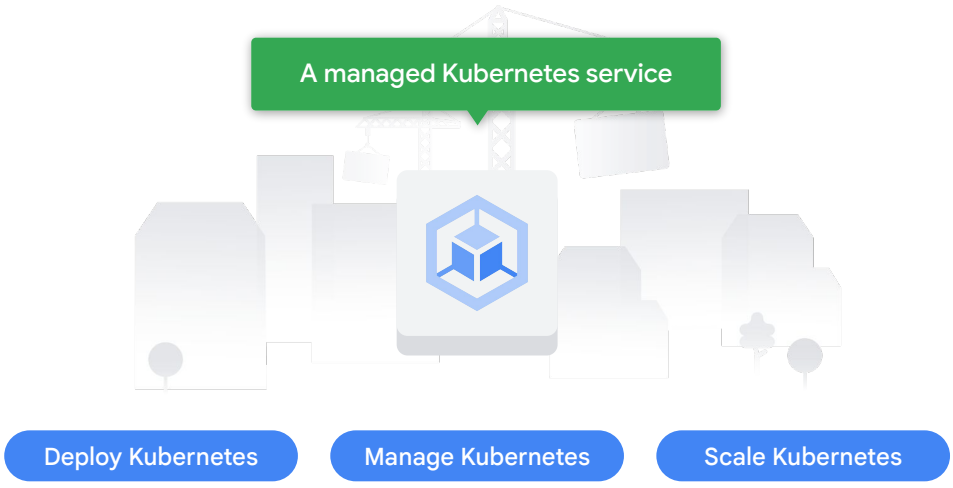
04 Google Kubernetes Engine



What if you started using Kubernetes, but the infrastructure is too much to maintain?

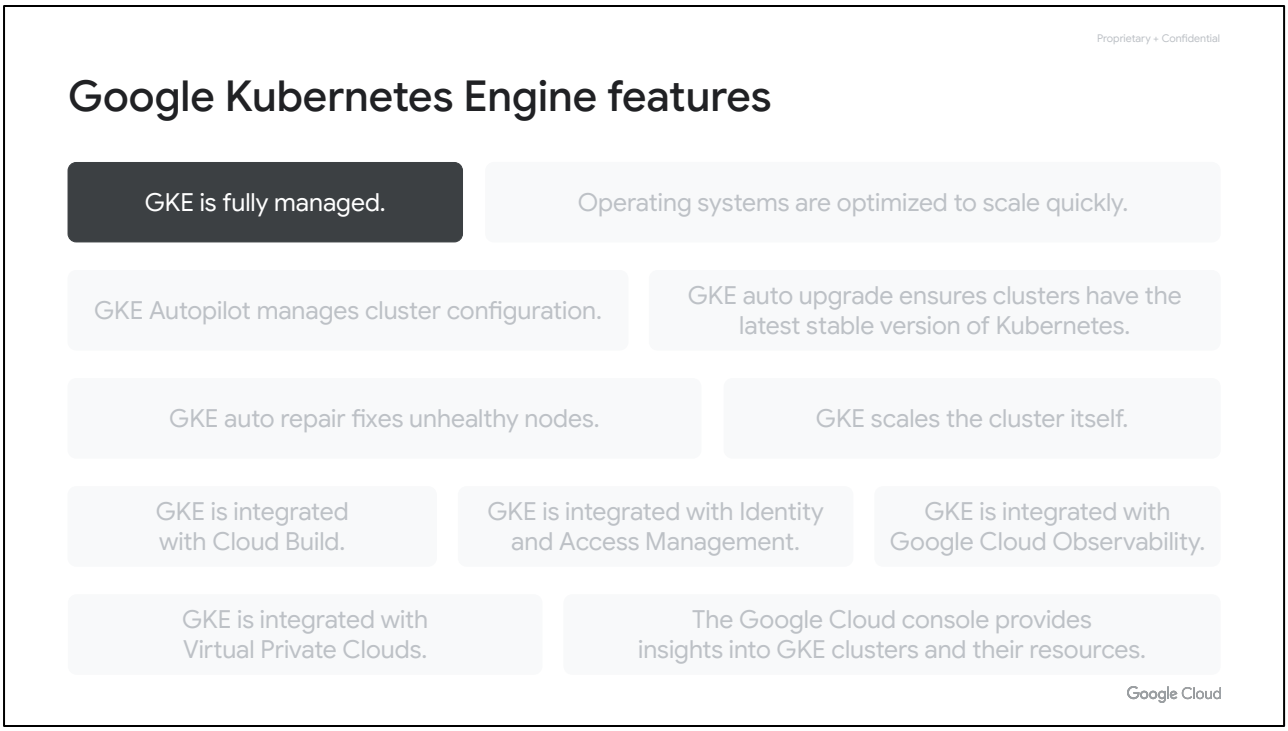
This is where Google Kubernetes Engine comes in.

# Google Kubernetes Engine

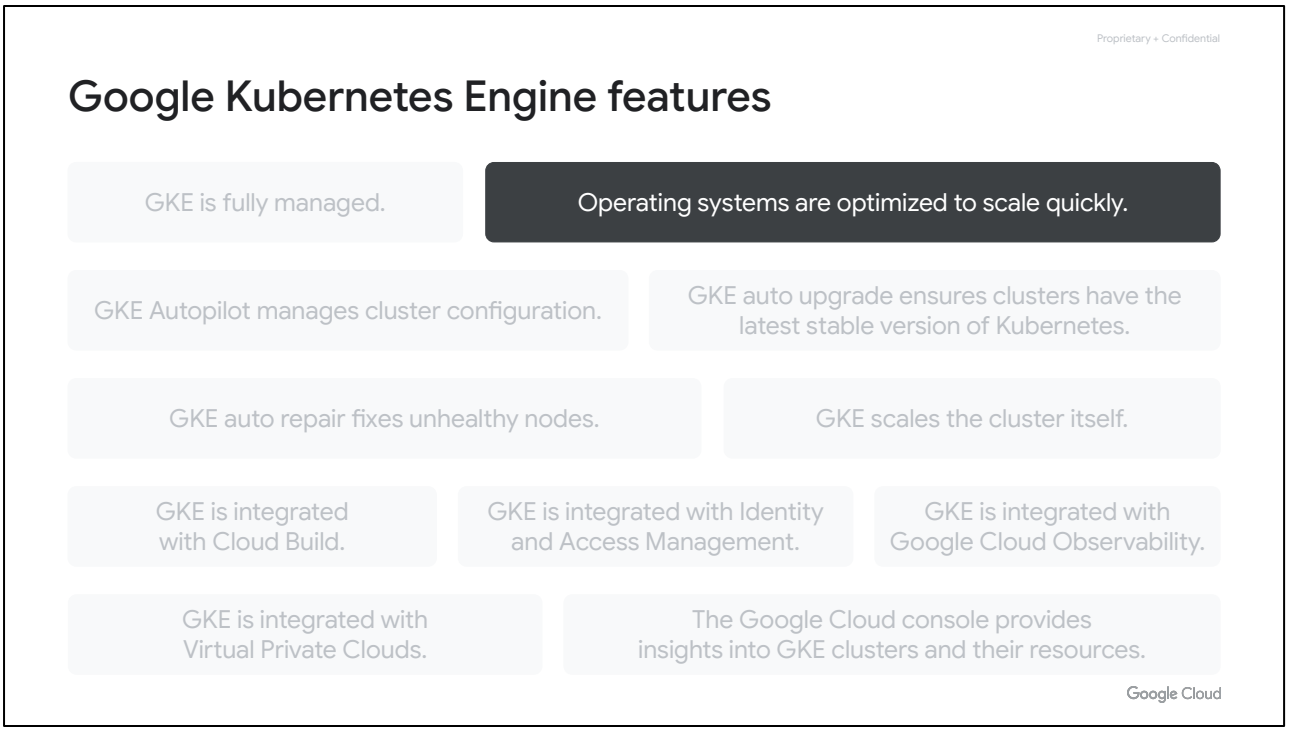




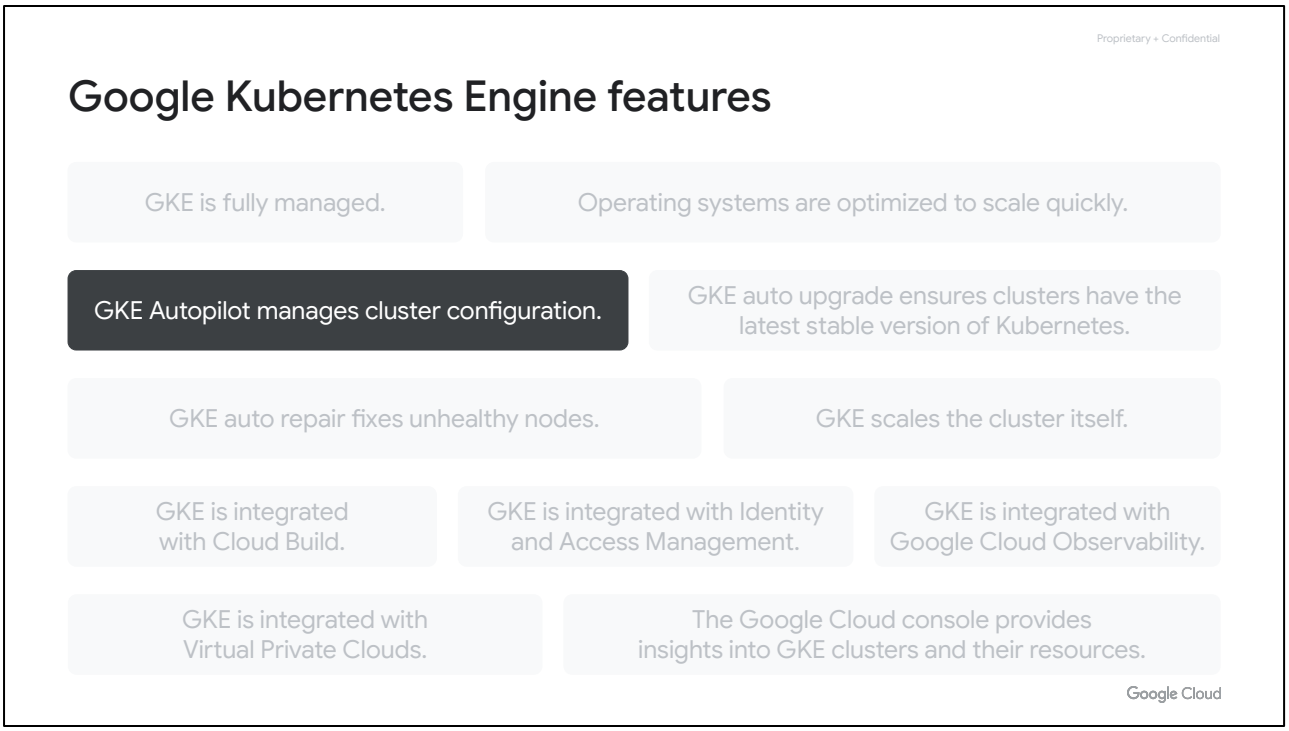

Google Kubernetes Engine is a managed Kubernetes service hosted on Google's infrastructure. It's designed to help deploy, manage, and scale Kubernetes environments for containerized applications.



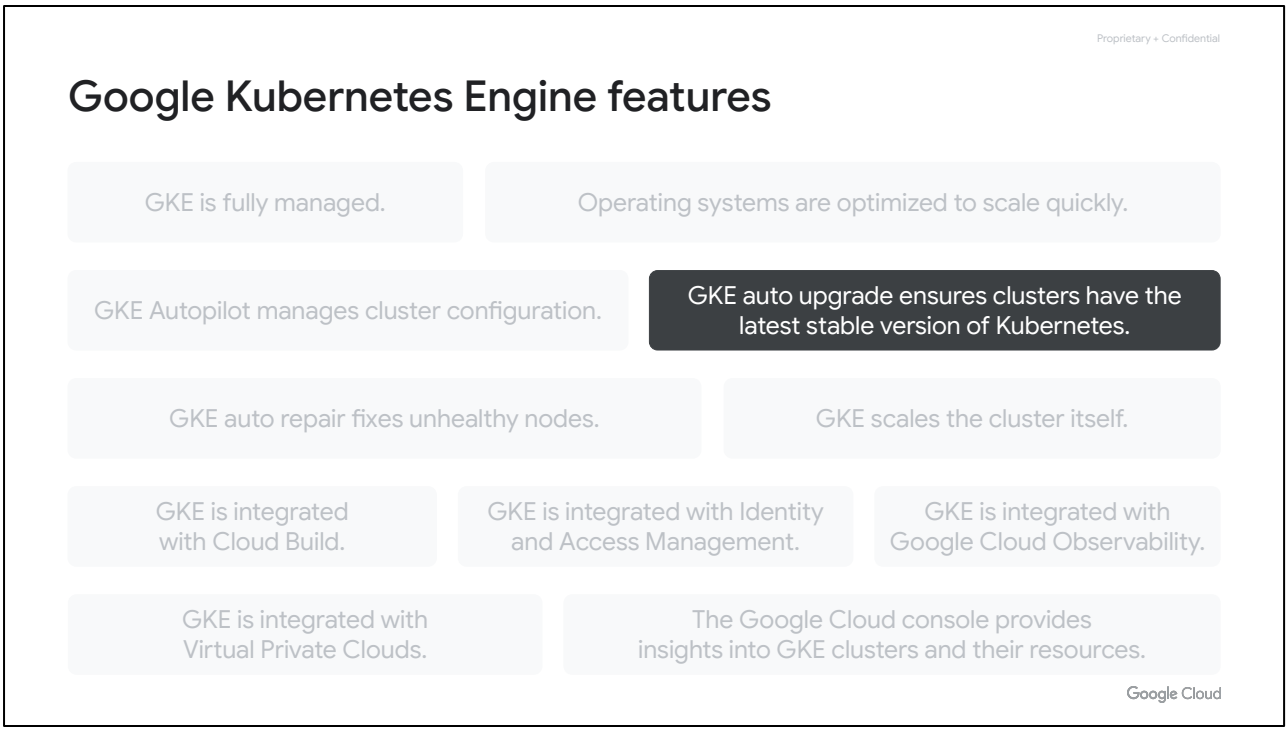

GKE is fully managed, which means the underlying resources don't have to be provisioned, and a container-optimized operating system is used to run workloads.




Google maintains these operating systems, which are optimized to scale quickly with a minimal resource footprint.

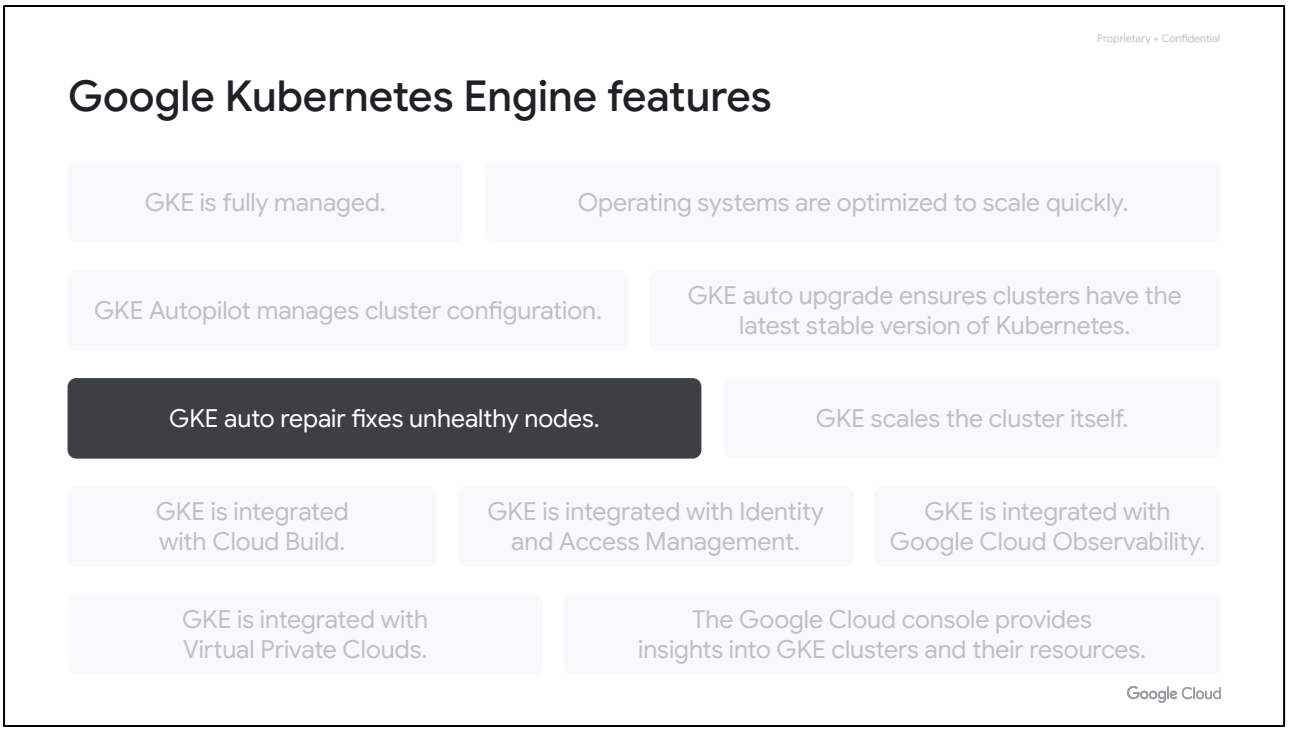



Google Kubernetes Engine offers a mode of operation called **GKE Autopilot**, which is designed to manage your cluster configuration, like nodes, scaling, security, and other preconfigured settings.

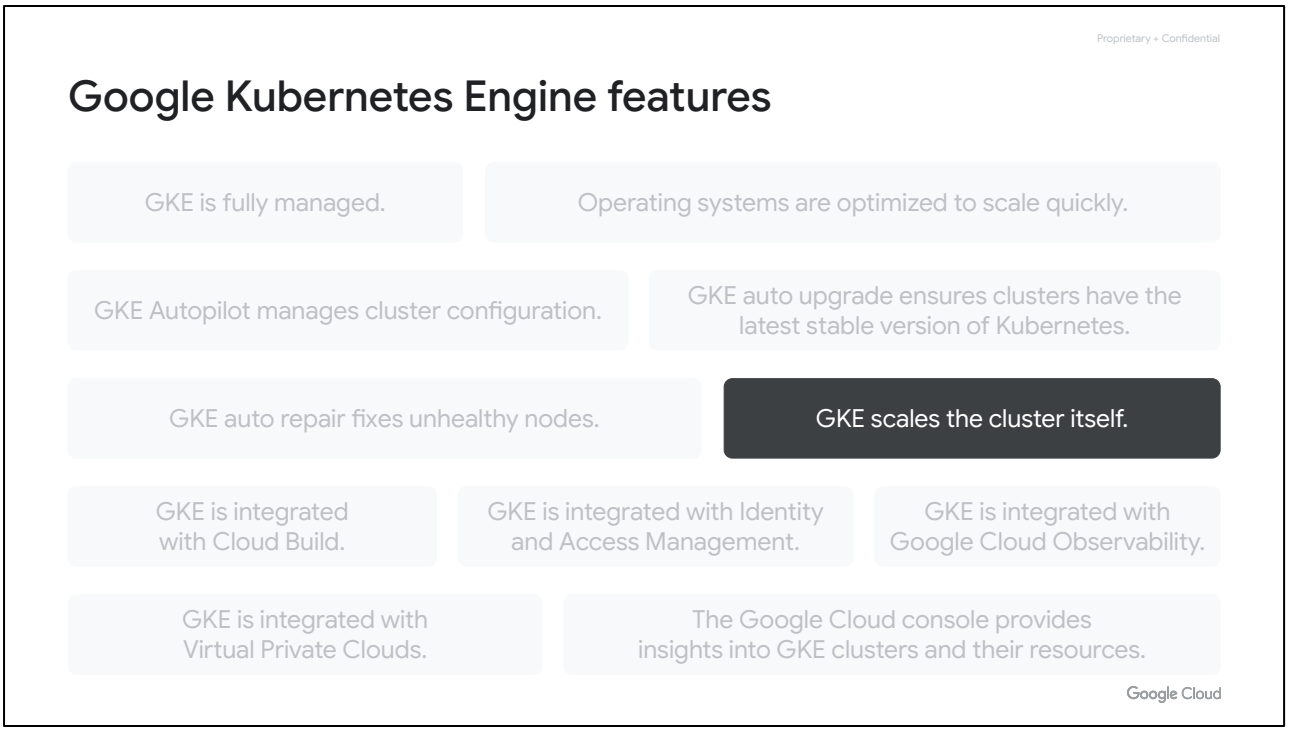



When you use GKE, you start by directing the service to create and set up a Kubernetes system for you. This system is called a cluster.

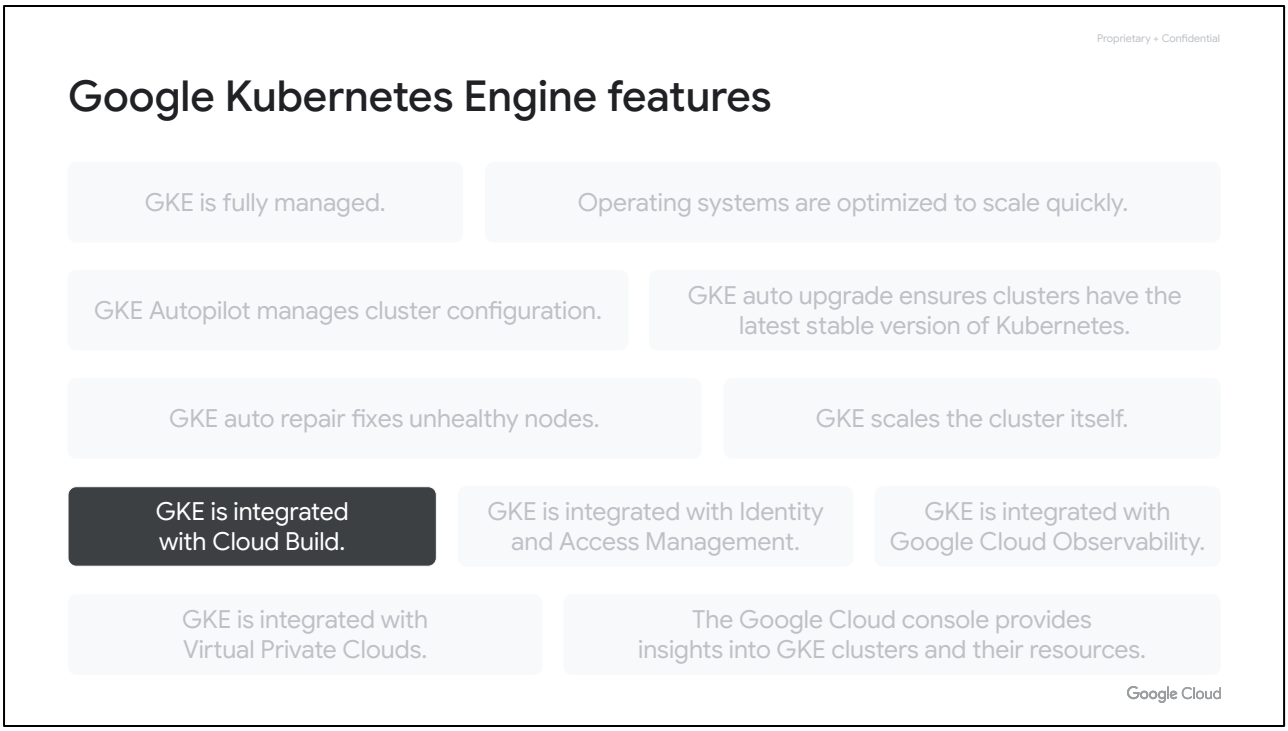
The GKE auto-upgrade feature ensures that clusters are always upgraded with the latest stable version of Kubernetes.




The virtual machines that host containers in a GKE cluster are called **nodes**. GKE has a node auto-repair feature that was designed to repair unhealthy nodes. It performs periodic health checks on each node of the cluster and nodes determined to be unhealthy are drained and recreated.

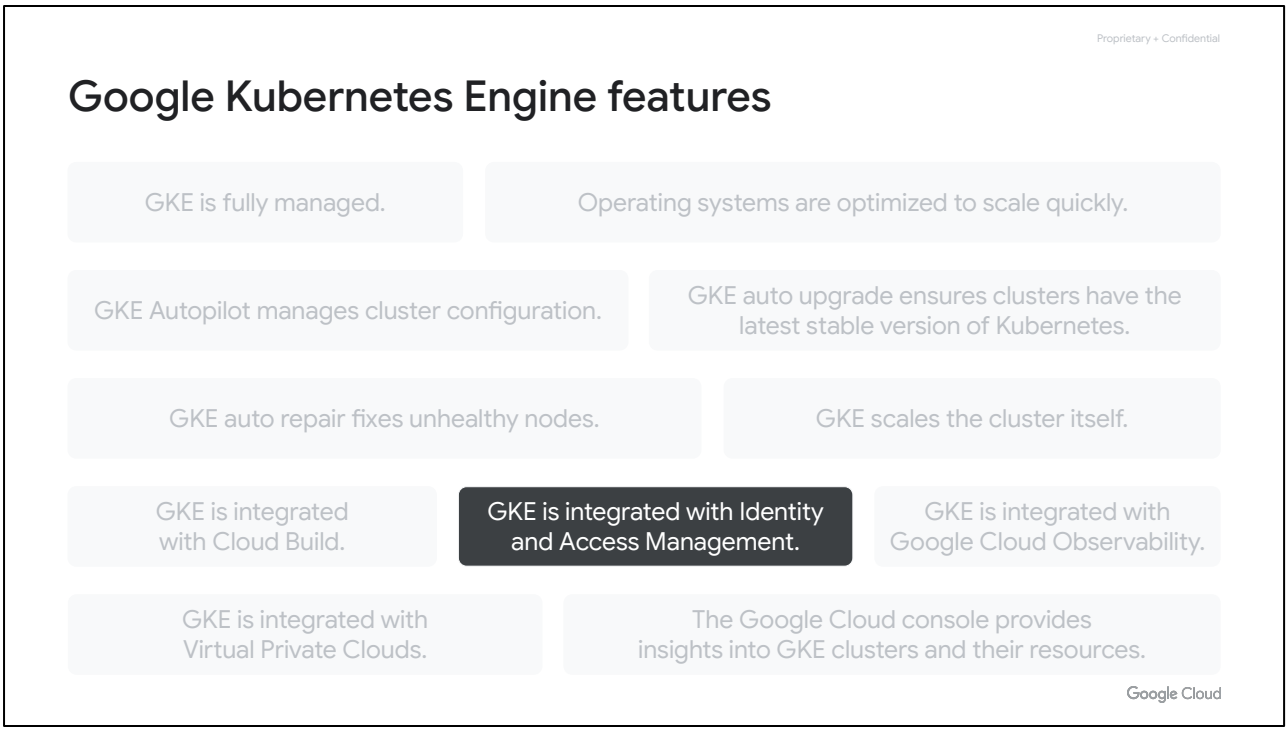



Just like Kubernetes supports scaling workloads, GKE supports scaling the cluster itself.

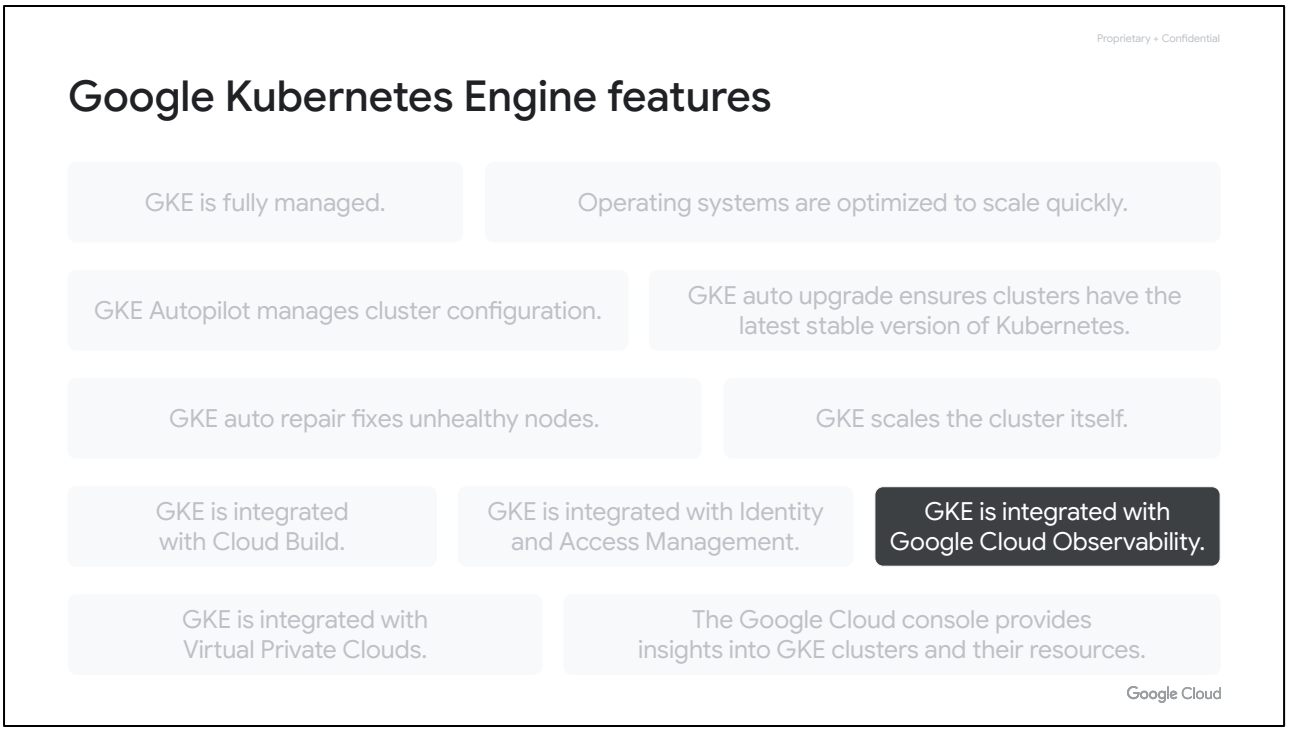



GKE is integrated with several services:

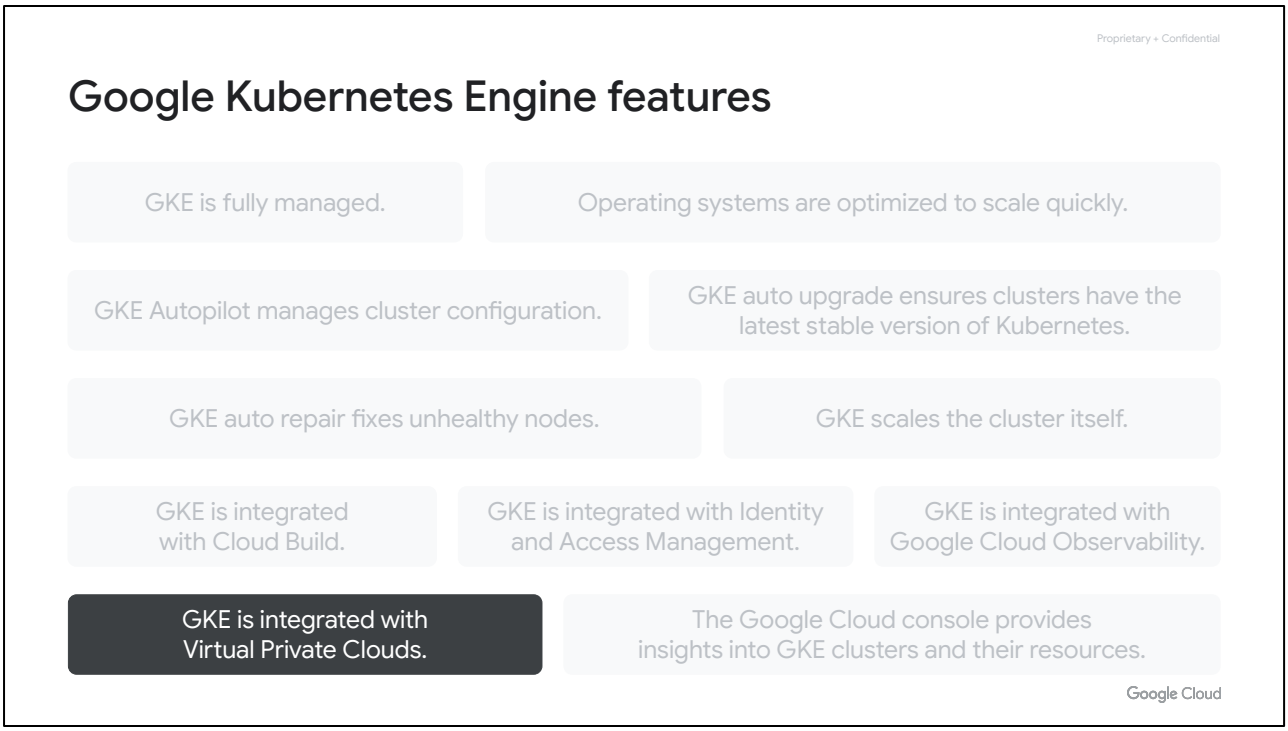
Cloud Build uses private container images securely stored in Artifact Registry to automate the deployment.



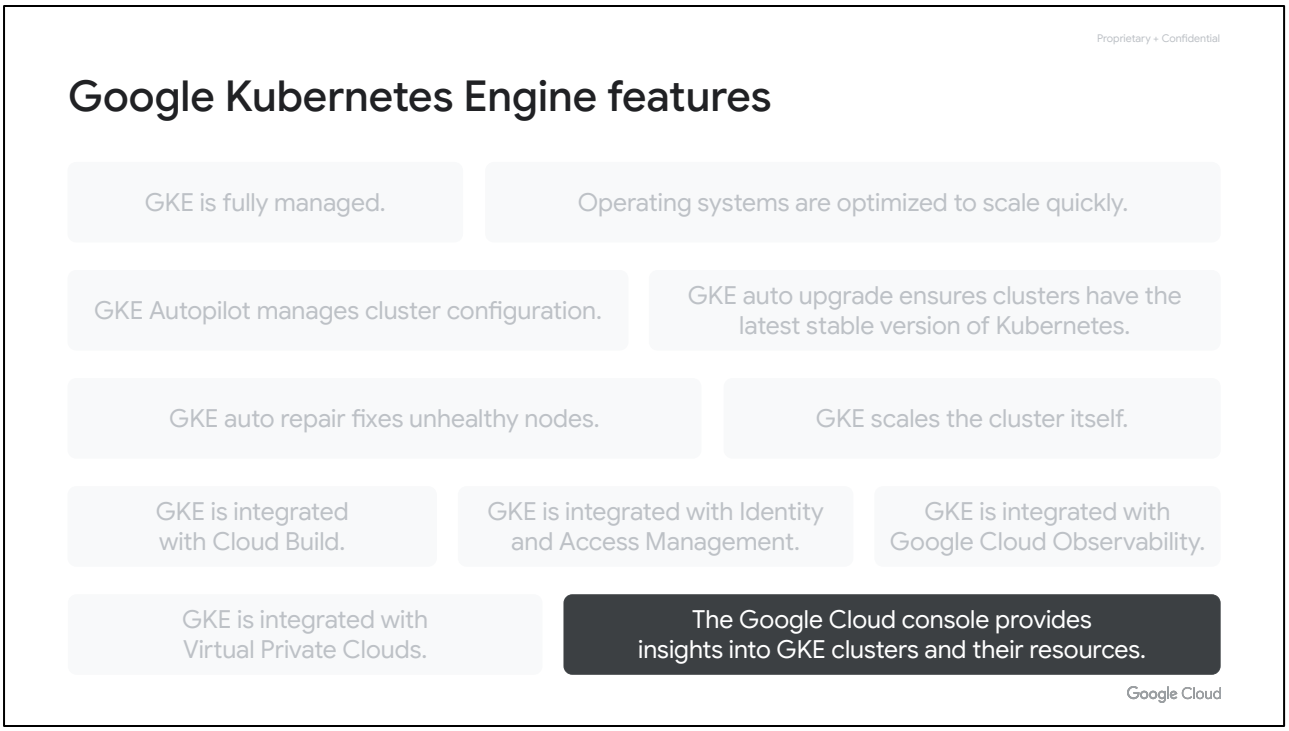

IAM helps control access by using accounts and role permissions.




Google Cloud Observability provides an understanding into how an application is performing.

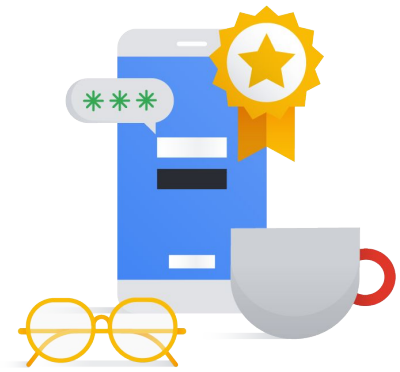



And Virtual Private Clouds, which provide a network infrastructure including load balancers and ingress access for your cluster.




And finally the Google Cloud console provides insights into GKE clusters and their resources, and a way to view, inspect, and delete resources in those clusters.

Although open source Kubernetes provides a dashboard, it takes a lot of work to set it up securely. However, with the Google Cloud console, there is a more powerful dashboard for your GKE clusters and workloads that you don't have to manage.



Time for a short quiz



Let’s pause for a short quiz.

# Quiz | Question 1 of 3

## Question

What is significant about the topmost layer in a container? Choose two options.

A. Reading from or writing to the topmost layer requires special privileges.

B. The topmost layer's contents are ephemeral. When the container is deleted, the contents are lost.

C. An application running in a container can only modify the topmost layer.

D. Reading from or writing to the topmost layer requires special software libraries.

# Quiz | Question 1 of 3

## Answer

What is significant about the topmost layer in a container? Choose two options.

A. Reading from or writing to the topmost layer requires special privileges.

B. The topmost layer's contents are ephemeral. When the container is deleted, the contents are lost. 

C. An application running in a container can only modify the topmost layer. 

D. Reading from or writing to the topmost layer requires special software libraries.

# Quiz | Question 2 of 3

## Question

When using Kubernetes, you must describe the desired state you want, and Kubernetes's job is to make the deployed system conform to that desired state and keep it there despite failures. What is the name of this management approach?

A. Imperative configuration

B. Virtualization

C. Declarative configuration

D. Containerization

# Quiz | Question 2 of 3

## Answer

When using Kubernetes, you must describe the desired state you want, and Kubernetes's job is to make the deployed system conform to that desired state and keep it there despite failures. What is the name of this management approach?

A. Imperative configuration

B. Virtualization

C. Declarative configuration 

D. Containerization

# Quiz | Question 3 of 3

## Question

What is the name for the computers in a Google Kubernetes Engine cluster that run workloads?

A. Nodes

B. Control planes

C. Containers

D. Container images

# Quiz | Question 3 of 3

## Answer

What is the name for the computers in a Google Kubernetes Engine cluster that run workloads?

A. Nodes 

B. Control planes

C. Containers

D. Container images